

# SOFTWARE FOR INTEGRATED MANUFACTURING SYSTEMS PART II

R A. Volz and A. W. Naylor  
The Robotics Research Laboratory  
The University of Michigan

July 28, 1987

## Abstract

Part I presented an overview of the Michigan unified approach to manufacturing software. This paper considers the specific characteristics of the approach that allow it to realize the goals of reduced cost, increased reliability and increased flexibility. It examines why the blending of a components view, distributed languages, generics and formal models is important, why each individual part of this approach is essential, and why each component will typically have each of these parts. An example of a specification for a real material handling system will be presented using our approach and compared with the standard interface specification given by the manufacturer. Use of the component in a distributed manufacturing system will then be compared with use of the traditional specification with a more traditional approach to designing the system.

This paper will also provide an overview of the underlying mechanisms used for implementing distributed manufacturing systems using our unified software/hardware component approach.

## 1 Introduction

Part I of this paper identified the following five concepts as the keys to our approach to manufacturing software:

1. Manufacturing software should be built in modern extensible general purpose languages.
2. Manufacturing software should be object oriented and created as assemblages of components.
3. Explicit formal semantic models are required.
4. Generics will amplify software reusability.
5. The above should be carried out in a largely distributed language environment.

In this paper, we explore the motivations for the use of these key concepts further and discuss an example of applying them to a material handling system.

The goals of our manufacturing software research are:

1. To develop techniques for building manufacturing software in a more reliable, less costly manner than present techniques.
2. To develop techniques for cost effective maintenance of manufacturing software.
3. To develop techniques for producing reusable software.
4. To develop techniques for producing portable software.
5. To develop techniques supporting a components industry.

Each of the five key concepts supports one or more of these goals.

## 2 Use of Modern Extensible General Purpose Languages

William Boller of Hewlett Packard<sup>1</sup> has recently stated, with respect to manufacturing software, that "Complexity is the root of all evil." Managing complexity is one of the most important things that must be done to develop reliable software. Managing complexity has also been one of the principal goals of software engineering research during the past two decades, and significant results have been obtained [1,2], including:

- Modular approaches to program development that provide a conceptually clear view of the system being implemented — This aids software production and maintenance.
- Powerful program verification techniques that, while not totally automatic — no existing technique is for programs of any size —, do automatically detect a very large fraction of program errors, thus reducing the cost of program development.

<sup>1</sup>"The Factory of the Future," *The Economist*, May 30, 1987.

- Modular approaches to program development that reduce compilation costs.
- Highly expressible and extensible capabilities.
- Portability of programs from one system to another.
- Techniques for managing concurrent/parallel real-time tasks.

Obtaining these same advantages for manufacturing software is important, and far more likely to be achieved a standard language is adopted than if a new one is built from scratch.

Among the language mechanisms used to achieve these results are:

- data encapsulation and hiding,
- data and program abstraction,
- strong typing,
- separate compilation (both of different modules and of module specifications and implementations), and
- explicit control of representations — particularly for numerics.

Placing all of these into a special purpose language is a very difficult, time consuming and error prone task. Yet omitting them would be to forgo some of the capabilities needed to achieve our goals [3,4,5].

### 3 Use Object Oriented Software

Our world is made up of objects, and we are accustomed to thinking about the management of our life in terms of the objects around us and operations that may be performed on them. E.g., I am editing on my computer terminal. I drive my car to work each day. Etc. It is natural to carry this mode of thinking over to our problem solving and system building activities, in which case, it is called object oriented design [6]. This approach helps develop a conceptual clarity of the system being built and organize its complexity.

When coupled with the representation of the object by a specification — the public interface to the object that presents the only ways (operations) by which the object may be accessed — and a body that implements the object, the object oriented approach is the natural mechanism for developing plug compatible components and a whole new approach to the relationship between suppliers and manufacturers.

With a components industry for manufacturing equipment and software in place, manufacturers would specify in a formal way the requirements for the manufacturing equipment they need and the component suppliers would supply manufacturing hardware and software components which would “plug” into the rest of the manufacturers system. Several things de-

rive from this view. First, the industrial manufacturer designs the package specification to provide the view of the manufacturing device necessary for the application at hand. Component suppliers are then given the compiled specification and must provide not only the required hardware, but a body to the component package which is compatible with the manufacturer compiled specification as well. Since the component is now formally specified and can be automatically machine checked for compliance with the specification, several vendors might bid against each other for the job. Second, since the body must reside in the control computer, the supplier must take responsibility for the applications level communication across the network. The supplied software component is directly pluggable into the manufacturer's computer. This is exactly the opposite of current practice in which the manufacturer assumes the responsibility for custom designing the hardware and software interfaces for integration of the system.

Third, since suppliers will have a fixed and standard framework within which they must deliver components, it will both be easier to develop custom products and easier to formulate standards when a class of devices has reached maturity.

### 4 Formal Models

We need models of the factory floor and process plans in order to develop control algorithms. Since we realize the factory floor and process plans as assemblages of software/hardware components, we are, in effect, concerned with formal semantic models for such components. The modeling methodology used is described in more detail in [7,8]. One component may include *models* of other components. The models may then be used in a predictive simulation manner to examine the likely outcome of a possible control strategy before it is actually applied.

Finally, the modeling methodology can be used to represent the process plans that the cell is to implement as well as the actions of the components. The uniform modeling of process plans and software/hardware components simplifies the software structure and allows one to view the process plans as just another component in the system. And, the formal models of process plans can be converted to actual components that drive the operation of the system. At present the translation from the formal models to actual software is performed manually, but conceptually (at present, and in the future actually) they could be converted automatically.

### 5 Generics

Generics can be used in a variety of ways. The most obvious was stated in Part I, to obtain software reuseability through what amounts to parameterization of the types and functions used in a component. However, generics can be used in other ways as well. They can be used to provide an individualized interface to a component, as will be illustrated below. That is, each user of a component, such as a material handling system,

can instantiate his/her own "view" or interface to the system. In this way, the interface to the system can be simplified.

One can also consider dynamic extensions to generics that would allow a user to create instances of generic components at run-time. In this case, each real component would contain the parameters necessary to complete a generic instantiation of it. The user would just reference the generic component and name a specific real component (for example a specific vehicle from a pool of vehicles in a material handling system) from which an actual instance of the component would be created. Resource managers, in particular, would find it useful to operate in this manner.

## 6 A Distributed Language Environment

Sec. 2 above described a number of advantages available from modern software engineering tools. These capabilities, however, are centered at the language level. That is, they are achievable for *single programs*. In the manufacturing world, however, we are clearly working across machine boundaries. Even for modest sized systems, there will be multiple control computers that will have to communicate with each other. In order to achieve the full advantages of modern software engineering, then, one should look to distributed program execution, that is, execution of a *single program* across a network of processors. One then obtains the advantages of conceptual clarity, modularity and automatic program verification currently possible with single programs on single machines. The single program view of a distributed system would allow verification to be done across the entire system instead of, as is now the case, only on the subsets of a program residing on a single processor. In addition, it reduces the programmer's view of interprocessor communication to interprocess communication, which is the programmer's natural view of communication; special application level communication protocols become unnecessary, and any lower level protocols become transparent to the programmer.

Our approach to this need has been to adopt a standard programming language intended for real-time operation and develop a distributed version of it. Because it is basically a good language, is subject to intense standardization efforts, and is ostensibly intended for distributed execution, we selected Ada. To achieve distributed execution, we have built a pre-translator that takes a single Ada program as an input and whose output is a collection of pure Ada programs, one for each targeted processor. This is somewhat akin to the way embedded SEQUEL is handled in the DB2 database management system.

Our distributed Ada system [9] allows us to distribute library packages and library subprograms statically among a set of homogeneous processors. We write a single program and use a `pragma` (essentially a compiler directive) called `SITE` to specify the location on which each library unit is to execute. For example, if a simple transport system were controlled by

computer number 2 and the cell control using it were on computer 1, a sample of relevant code might look as follows:

```
pragma SITE (2);
package VEHICLE is
    procedure MOVE_FORWARD;
    :
end VEHICLE;
:
pragma SITE(1);
with VEHICLE;
procedure CONTROL is
    :
begin
    :
    VEHICLE.MOVE_FORWARD;
    :
end;
```

Our translation system would replace the local call to the procedure `VEHICLE.MOVE_FORWARD` with the appropriate remote call. Similarly any references in `CONTROL` to data objects defined in package `VEHICLE` would be translated into appropriate remote references as would task entry calls. Note that the user need only use the normal procedure call mechanism to cause the vehicle to move.

## 7 Material Handling System Example

[7] describes a generic factory control system that has been built and simulated using the ideas described above. In this section, we explore one component of such a system in more detail, a material handling system.<sup>2</sup> We suppose a material handling system (MHS) that is used to move pallets from one location to another, has a number of vehicles to carry out the moves, and can be utilized by several different parts of the system.

From a hardware/software component, i.e., object, perspective we think of the *relevant* objects in the system and the functions performed on them *by the parts of the system that need to use the MHS*. In the simplest view of this example, the relevant objects (from the perspective of the user of the MHS) are the *MHS* itself, the *pallets* that are to be moved, and the *locations* to/from which they pallets are moved. The vehicles used are not relevant to the user, and thus should remain hidden from the view provided to the MHS user. Since there are potentially multiple parts of a factory system, e.g., multiple cells, that could have need to more or less independently make use of the MHS, the MHS should support a concept of multiple users. However, for any one user, the view of the MHS should not have to be cluttered with unnecessary detail about the other users. Generics allows us to achieve this.

<sup>2</sup>We have actually implemented a more complete abstraction of a material handling system than that described here.

We show here a simplified (only in the sense of a reduced set of operations supported by the MHS) generic interface to the component:

```
generic
package GENERIC.MHS is
  type PALLET is private;
  type LOCATION is private;
  L1,L2,L3,L4,LN: constant LOCATION;
  type MOVE.ID is private;
  type ACKNOWLEDGE is (OK, BUSY, FULL);
  type MOVE.STATUS is (WAITING, MOVING, DONE);
  MHS_NONRESPONDENT: exception;
  procedure ALLOCATE.PALLET(P: out PALLET; ACK: out ACKNOWLEDGE);
  function WHERE.PALLET(P: PALLET) return LOCATION;
  procedure REQUEST.MOVE(P: PALLET; L: LOCATION;
    M: out MOVE.ID; ACK: out ACKNOWLEDGE);
  procedure MOVE.STATUS(M: MOVE.ID; MS: out MOVE.STATUS);
private
end GENERIC.MHS;
```

A cell controller using the MHS might look something like the following:

```
pragma SITE(1);
with GENERIC.MHS;
procedure CELL.CONTROL is
  package LOCAL.MHS is new GENERIC.MHS;
  use LOCAL.MHS;
  P1, P2: PALLET;
  MS: MOVE.STATUS;
  M1, M2: MOVE.ID;
  ACK: ACKNOWLEDGE;
begin
  ALLOCATE.PALLET(P1,ACK);

  MOVE.REQUEST(P1, L1, M1, ACK);

  ALLOCATE.PALLET(P2,ACK);

  MOVE.REQUEST(P2, L2, M2, ACK);

  MOVE.STATUS(M1, MS);
end CELL.CONTROL;
```

There are a number of points to notice about this example. First, the instantiation of the generic MHS provides a clear and straightforward interface to the MHS, expressed in terms a user would find convenient in dealing with the MHS component. The command names have been chosen to have an implied semantics indicative of the operation to be performed. Reading the control program is straightforward. The types provided are just those needed to talk about the objects associated with the MHS. Irrelevant details are hidden.

This example is also presented in terms of a distributed system. The cell controller is indicated as being located on site 1. It is not stated where the MHS is located, and the only fact about its location that is relevant to the cell controller is the fact that it might be on a different computer. In this case, the function and procedure calls to the MHS object will involve remote calls to the site at which the actual MHS controller is located. This possibility is manifested in

the generic MHS through the exception NON\_RESPONDENT. When the user (CELL.CONTROL in this case) instantiates a copy of GENERIC.MHS, that copy will appear on the same computer as CELL.CONTROL. Hidden in the implementation of the local copy, LOCAL.MHS, is a periodic checking of the communication line and a timeout on the return from the remote procedure calls. If the communication line fails or the actual MHS does not respond within its prescribed time, the implementation of LOCAL.MHS will raise the exception NON\_RESPONDENT, and CELL.CONTROL can deal with this as necessary. Only the abstraction representing failure of the actual MHS is appropriate for CELL.CONTROL to be concerned with; of course, other kinds of failures could equally well be represented.

The translation system supporting distributed program execution replaces all calls to remote components with the appropriate communication routines and implicitly manages communication routing.

Also note that by focussing on an object oriented view of the components the potential for standardization is increased. It is now easy to think in terms of standardizing the interface to a single component type, such as an MHS, without having to consider any other component types in the system. The types, procedures, functions, exceptions and call profiles become the formal expression of the standard. Moreover, the syntactic compliance to a standard can be automatically checked by the system compiler.

## 8 Conclusion

A coherent approach to manufacturing software is one of the most important building blocks needed for U.S. industry to truly develop integrated manufacturing systems. We have described a concept by which coherent manufacturing can be accomplished. However, the theory is not yet complete. Indeed, much remains to be done. Extensions to the formal modeling system are needed to more fully handle generics and distribution of components. The process of instantiation of generics to real components must be extended to allow dynamic instantiations. Distributed languages must be studied in a more general context of multiple forms of memory interconnections, multiple possible binding times, and various degrees of homogeneity (e.g., see the major dimensions of a distributed language defined in [10]).

Yet, we have accomplished enough to demonstrate the viability of the major underlying ideas. A primitive version of a distributed Ada translation system is working, and a limited generic real-time factory controller is operational, with real factory components replaced by simulation. We believe that when it is fully developed, the approach presented here can become the heart of future integrated manufacturing systems.

## References

- [1] B. Liskov and J. Guttag. *Abstraction and specification in program development*. MIT Press, Cambridge, MA, 1986.
- [2] J.C. Cleaveland. *An Introduction to Data Types*. Addison-Wesley, Reading, Mass, 1986.
- [3] E. Denert. *Trends in Information Processing Systems. 3rd Conference of the European Cooperation in Informatics*, chapter Software Engineering: Experience and Convictions, pages 16–35. Springer-Verlag, October 1981.
- [4] S.N. Woodfield, H.E. Dunsmore, and V.Y. Shen. The effect of modularization and comments on program comprehension. In *5th International Conference on Software Engineering*, pages 215–23, March 1981.
- [5] L. Varga. Specifications of reliable software. *Tanulmanyok Magy. Tud. Akad. Szamitastech. And Autom. Kut. Intez. (Hungary)*, (113):309–25, 1980.
- [6] Grady Booch. *Software Engineering with Ada*. Benjamin/Cummings, second edition, 1987.
- [7] A.W. Naylor and R.A. Volz. Design of integrated manufacturing system control software. *IEEE Trans. on Sys., Man, and Cybernetics*, submitted 1987.
- [8] A.W. Naylor and M.C. Maletz. The manufacturing game: a formal approach to manufacturing software. *IEEE Trans. on Sys., Man, and Cybernetics*, SMC–16:321–334, May-June 1986.
- [9] R.A. Volz, P. Krishnan, and R. Theriault. An approach to distributed execution of Ada programs. In *NASA Workshop on Telerobotics*, to appear 1987.
- [10] R.A. Volz, T.N. Mudge, G.D. Buzzard, and P. Krishnan. Translation and execution of distributed Ada programs: is it still Ada? *IEEE Transactions on Software, Special Issue on Ada*, to appear 1987.